*Implementation of Actor Model*

*Abstract*—To take the benefit of hardware concurrent computational models is growing rapidly. The advancements in cloud computing technologies such as Microsoft Azure Service Bus allows improving the computational speed. This project illustrates how the Actor model system is implemented on top of the Microsoft Azure Service Bus. A client-side implementation is also provided in this paper to test the functionality.

**Keywords— *Concurrency, Microsoft Azure Service Bus, Actors, Threads, Topics, Queues, Azure Cosmos DB***

# **Introduction**

In this era of Cloud computing and Distribution systems concurrency has played a vital role in achieving fast results with low latency. It is possible only if the code runs concurrently. Concurrency is a property of the system to execute multiple activities at the same time. It means how components should work in a concurrent computational environment.

The actor model is a conceptual concurrent computation model, came into the picture in 1973[1]. It establishes some of the rules on how the system’s components should behave and interact with each other [3]. An Actor in the Actor model can be represented as a fundamental unit of computation and it can perform actions such as create another actor, send a message and designate how to handle the next message. Actors are lightweight and millions of them can be created very easily. Also, it is important to note that it takes fewer resources than Threads. An Actor has its private state and a mailbox, like a messaging queue. A message which an actor got from another actor is stored in the mailbox and is processed in FIFO (first in and first out) order. Actors can be considered as the form of object-oriented programming which communicates by exchanging messages. Also, actors have a direct lifecycle that is they are not automatically destroyed when no longer referenced, and once created it is a user’s responsibility to eventually terminate them. This enables the user to control how the resources are released. Furthermore, in the case of distributed environments actors can communicate with each other through messages if they have the address of other actors, Actors can have local or remote addresses [1]. The most widely used implementations for the Actor model are Akka and Erlang [1].

The main inspiration behind Actor Model is to take full advantage of the hardware by using concurrency. Concurrency means that the ability of the system to perform different tasks simultaneously or out of order without affecting the outcome. DotNetActors is an essential element in the Actor Model and is responsible to handle the communication between the client and Microsoft’s Azure Service Bus. Following is a brief description of the different Microsoft Azure storages that are used to implement the whole infrastructure.

### Requirements for Actor Model: To run the Actor Model system using Microsoft Azure Cloud, knowledge of the following elements is required.

* 1. *Azure Table storage:* Azure Table storage or Cosmos DB Table API serves as storage for structured data in the cloud with schema-less design. This makes it easier to adapt the data as per the requirement of the application. Table storage can be utilized for storing flexible datasheets and other types of metadata as per the service requirements. Table storage is consisting of a Storage account, storage table, and entities, as shown in Fig1. [4]

**Diagram

Description automatically generated**

Fig. 1. Hierarchy of Table storage

* 1. *Azure Service Bus:* “Microsoft Azure Service Bus is a fully manages enterprise message broker with message queues and publish-subscribe topics” [5]. It provides many benefits such as load balancing the load across different workers, transferring the data safely, and coordinating transactional work. The service bus is a platform as a service (PaaS) with an additional feature that Azure takes care of such as Logging, managing space, handling backups, worrying about hardware failure, etc. The main protocol used is Advanced Messaging Queuing Protocol (AMQP) 1.0. In a Service Bus, Namespaces are the containers for all the messaging components. Multiple Queues and Topics can be in a single namespace. “A Service Bus namespace is your capacity slice of a large cluster made up of dozens of virtual machines” [5]. Due to this, it provides all the availability and robustness benefits on a very large scale.
  2. *Queues (Azure Service Bus):* Queues can send and receive messages. Messages are stored in the Queues until the receiving application has received and processed them.

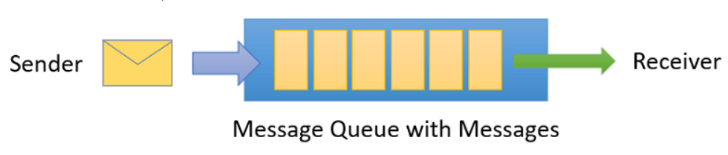
****

Fig2. Microsoft Azure Queue [5]

On arrival, the Messages in Queues are ordered and timestamped. It is spread across availability zones once accepted by the broker. It is always held in triple-redundant storage. Once the client has accepted the message Service Bus never leaves the messages in memory or volatile storage. Messages are only delivered when requested, which means they are delivered in pull mode. A pull model is not like busy-polling mode, the pull operation can be long-lived and gets completed once the message is available.[5]

* 1. *Topics (Azure Service Bus):* Topics can also be used to send and receive messages. As stated above Queues are used for point-to-point communication, Topics on the other hand are used in publishing/subscribe scenarios.[5]

**Table

Description automatically generated**

Fig3. Microsoft Azure Topic [5]

The key difference is that multiple and independent subscriptions can attach to a Topic, apart from this it works exactly like a Queue. Copy of each message that is sent to a topic is received by a subscriber. “Subscriptions are names as entities” [5]. A set of rules can be defined in a subscription known as a filter which can keep a check on which message to be copied to a subscription and an optional action that can modify the message metadata [5].

* 1. *Containerization of the Actor Model using Docker:* Containerization is a time reducing and cost-effective way to achieve rapid App deployment, as no hardware configurations and software installations are required to host a deployment. The principle purpose of containerization is to let multiple apps run in single hardware inside their isolated containers, without interfering with files, resources, memory, and processes utilized by other running apps. To fulfill these requirements, Docker serves as the best solution, through which an app can be quickly deployed and run in its designed environment, both locally and on the cloud. The app first needs to be packed up as a Docker image, which can be tested locally by using Docker for Windows, or on could by using Microsoft Azure Container Registry and Instance service [5].

The paper is structured as follows; Section II covers the general workflow. Section III explains the Implementation of the Actor Model System. Results and conclusion are presented in sections IV and V respectively.

# **General Workflow**

The main objective of this paper is to implement the Actor Model on top of Microsoft Azure Service Bus using Microsoft’s official documentation. The project workflow is illustrated in Fig.4

Diagram

Description automatically generated

1. Client creates an Actor and calls the Ask Method.

2. Service fetches the message.

3. Persisting the Actor in a Table

4. Output is sent to the Queue

5. Client fetches the output from the Queue.

Fig4. General Workflow

Following are the implemented steps according to the above-illustrated workflow, to achieve the goal of this project.

1. *Implement a Sample client:* Firstly, a sample client is implemented and the connection with the Azure service bus is established. This Client then initializes the Actor System and creates an Actor. After this Actor Reference is configured to this Actor which calls the *Ask*() method and sends a message to the Topic. It then starts polling the Queue to fetch the output.
2. *Service Subscribes the Message from Topic*: The service then subscribes the message from the Topic. The subscribed message is then processed by the service.
3. *Persisting the Actor in a Table*: If the persistence is enabled in the Service then the Actor is persisted in the CosmosDB table.
4. *Uploading output to the Queue*: Service uploads the output to the Queue.
5. *The client fetches the output from the Queue:* The output is fetched from the Queue and is sent to the client.

# **CODE DESCRIPTION**

# *Dotnet Actor Service Host Code*

The execution of this project according to the workflow illustrated above is implemented by the class Program.cs. It is situated in the DotNetActorsHost solution in the project folder as shown in Fig.5

Graphical user interface, text, application

Description automatically generated

Fig5. Code Structure

In the Program.cs file there is the main method that starts the Host application. It is important to note down that the Service Bus connection string needs to be set up as an environment variable as shown in Fig6

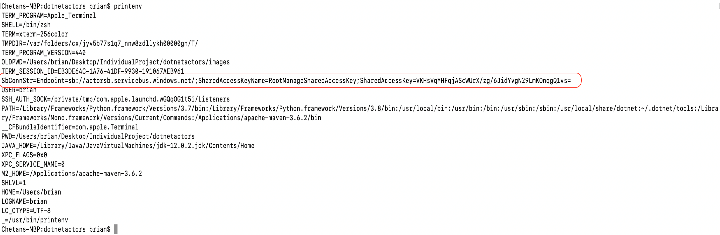


Fig6. Environment Variables

Also, command-line arguments that are mandatory run the program are:

* *SystemName*: Initializes the Actor system with this name.
* *RequestMessageTopic*: Topic from where the system will subscribe the client messages.
* *RequestMsgQueue*: Queue where the output results are stored and are made available for the client.
* *ActorSystemName*: This name is used in building the name of the Azure table.
* *SubscriptionName*: It represents the subscription name from where the service should subscribe its messages.

The main program that starts the Actor Host service is shown in Fig7

Graphical user interface, text, application, email

Description automatically generated

Fig7. Main Program in DotNetActorHost Solution

As highlighted in Fig ?? an *ActorSbHostService* object is created and a logger is passed as an argument to enable logging. In the next line, the object calls the method *Start*() Fig of Start method in class ActorSbHostService

and passes the command line arguments (stated right above) in it. The *Start()* method is implemented in the class ActorSbHostService.cs class as shown in Fig8.

****

Fig8. Code snipper of Start Method in ActorSbHostService class

In the first line of the code, the ActorSbConfig object is initialized. This object takes in all the configuration that is required to initialize an Actor system. The configurations such as *SbConnStr, TblStoragePersistenceConnStr, RequestMsgTopic, ActorSystemName* and *requestSubscriptionName* are configured in this method. Going further down as shown in fig9 a table is initialized using the table connection string.

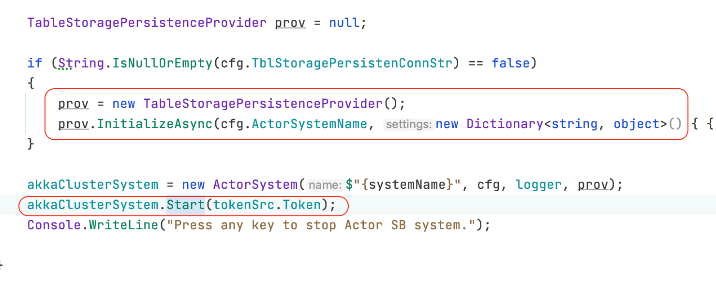
****

Fig9. Code Snippet for Table Persistence

After the table initialization is successful, the ActorSystem object is initialized. The ActorSystem calls the Start() method as shown in Fig10.

****

Fig10. Code snippet of start method in ActorSystem class

In Fig10, two tasks are created to perform the actions asynchronously. The first task creates a SessionClient and subscribes the messages from the Topic asynchronously whereas the second task waits for a cancellation token in case the program needs to stop. The method *RunDispatcherForSession()* is called if a message is there in the Topic.



Fig11. Code snippet of Task1 implementation

As shown in Fig11 a message is subscribed from the topic. Then a not equal null check is validated on the subscribed message. After that Message type is checked and a null pointer exception is thrown if the type is null. In the next line, it is checked if the actor is there in the actorMap using the key SessionId. It will return value if the actor was persisted. After that, an Actor is loaded using the persistence provider object. In case the actor is not found it is stored in actorMap using its sessionId.

****

Fig12.a Code snippet of RunDispathcherForSession

Then the message is deserialized using the method *DeserializeMsg()* method. This is done to perform the necessary operations on the received message. In the next step *InvokrOperationActorAsync()* is called to operate on the invokingMessage. This performs the required operations on the deserialized message.



Fig12.b Code snippet of RunDispathcherForSession

In the *InvokrOperationActorAsync()* method an internal method named *Invoke()* is called and deserialized message is passed as an argument. This method returns a response which is then further serialized to create a response object that can be sent over to the service bus.

****

Fig13. Code snippet of InvokeOperationOnActorAsync

The response message is created using the *CreateResponseMessage()* method as shown in Fig13

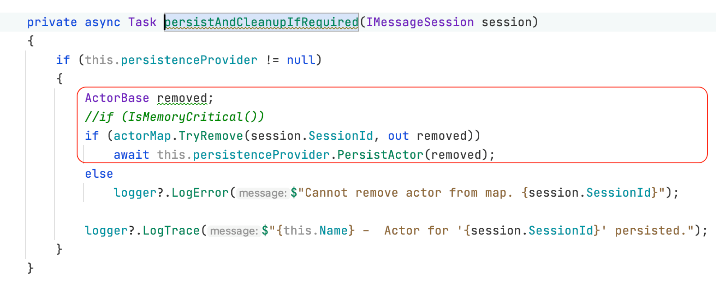
****

Fig14. Code snippet of PersistAndCleanupIfRequired method

After this, if persistence is required the actor is persisted in the cosmos DB Table using the method *persistAndCleanupIfRequired().* In our case we have a Table persistence provider which is invoked by calling PersistActor() method as shown in Fig14.

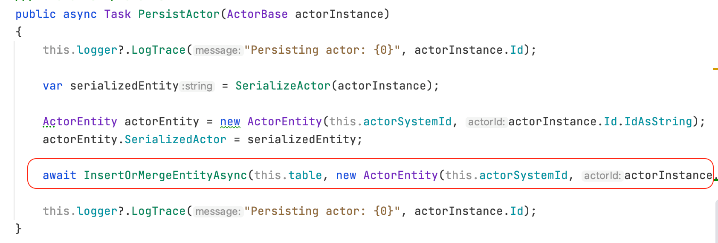
****

Fig14. Code snippet of PersistenceActor method

In the next step the if the reply is requested the response message is sent to the Service Bus Queue using the method *sendReplyQueueClients().*

# *Sample Client Code*

A sample code has been implemented to test the functionality of the Actor model. Below is the code description of the implemented code. The name of the solution is *CarSampleActor*. It consists of one single class named Program.cs as highlighted in Fig15.

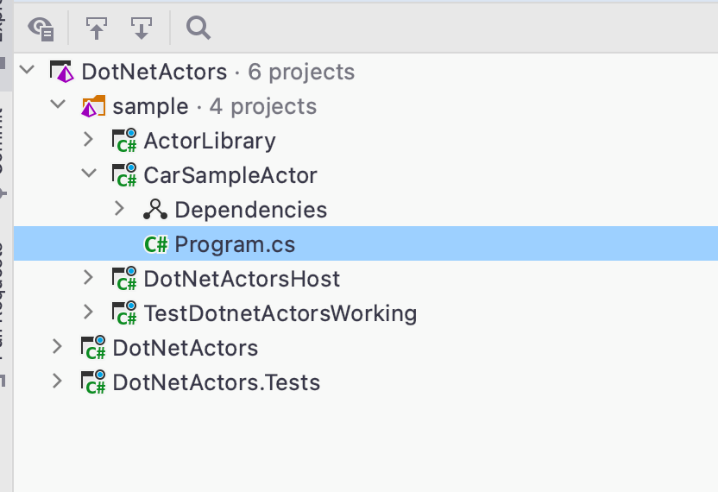


Fig15. Code Hierarchy of the client

The main method creates the object of *ConfigurationBuilder* class. After that, the object builds the command line arguments and the environment variables as *configures*. One of the arguments that need to be passed before running this project is *shallRun***.** It can be set as true or false. If kept true it calls the method *LoadCarAttributes*() whereas if kept false it calls a different method named as *CheckPersistence*() method as shown in Fig16.

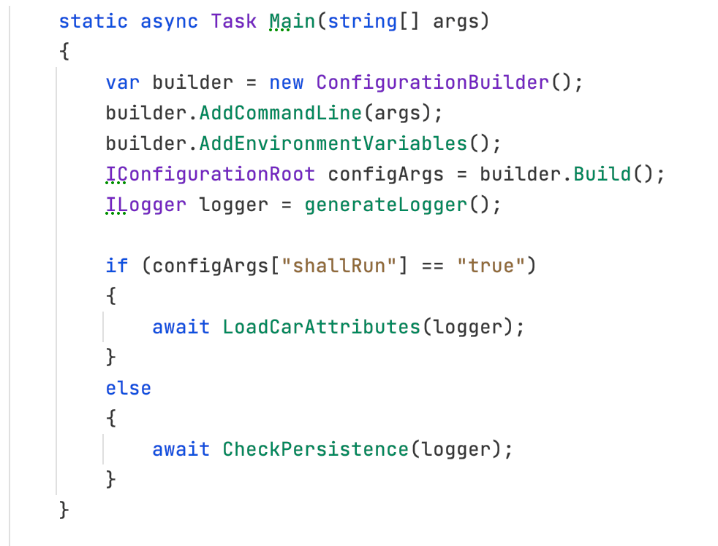
****

Fig16. Code snippet of the main method in Client Sample

*LoadCarAttributes*() method is used to pump around 100 Actors into the service. In the first step, a *CancellationTokenSource*() object is created that can be used to terminate the program. In the next step, Configurations are fetched by calling the method *GetLocalSysConfig*() method. This method returns the configurations such as *SbConnStr*, *ReplyMsgQueue*, *RequestMsgTopic* and *ActorSystemName*. As stated above all these configurations are used to create an ActorSystem.

****

Fig17. Code snippet to get Local Configurations

After initializing the ActorSystem object as shown in Fig?? a for loop is used to perform 100 iterations that create Actors by calling *ActorSystems* *object.CreateActor*() method. In each iteration and calls the ASK() method and passes the arguments such as car attributes as “green”, carSpeed as “222 ­+ i ” where i is the iteration number. The response from these *ASK*() methods is then logged using a logger. The method is shown in Fig18.

****

Fig18. Code snippet to Load Car Attributes

After the successful execution of the above method now the main program is again called but this time the command line argument is kept as false. This will invoke *CheckPersistence*() method. In this method first step is to get the configurations by calling the *GetLocalSysConfig*() as shown in the above fig??. In the next step object of ActorSystem class is created and configurations are passed to its constructor as one of the arguments as shown in the below fig19.

****

Fig19. Code snippet to Check the persistence

After that, a for loop starts that iterate for 100 cycles, and in each iteration an Actor is created with id ‘i’ and each of these Actor calls an ASK() method and passes ‘i’ as its argument. The response from this method is the speed of the car.

# **IV. Results**

The extensive stream of methods discussed in this paper takes much less time on accounts of running the experiment. It's due to the robustness of the Microsoft Azure Service bus and Actor Model System. The following images illustrate the run of the code.

Fig20 shows when the service is running

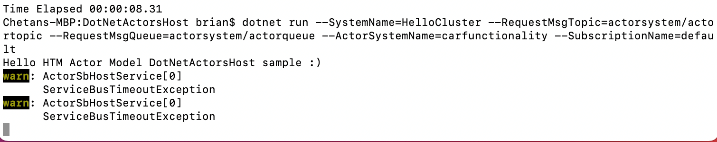


Fig20. Actor Host Service running

Fig21 shows when the car sample run by passing command-line argument *shallTrue* as true.

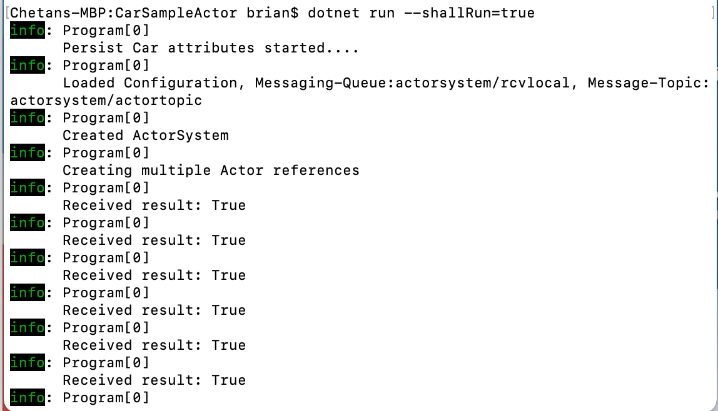


Fig21. Sample client solution running

Fig22 represents the service logs when receiving messages from the client.

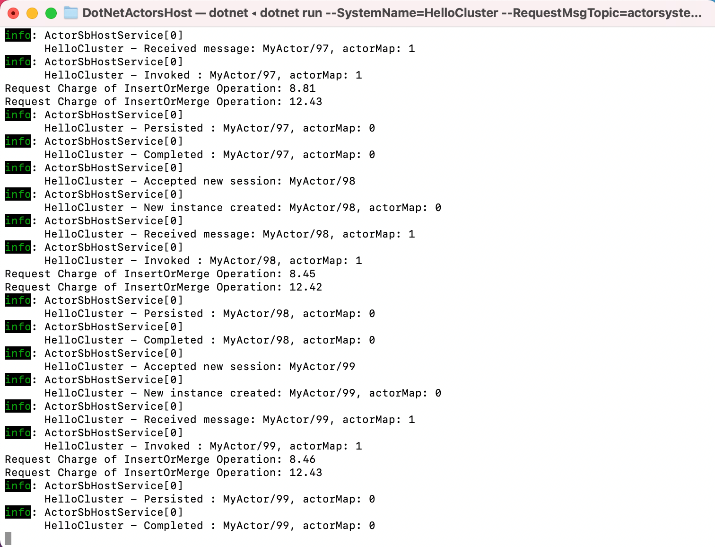


Fig22. Dotnet Actor Host Service Logs

Fig23 shows the persisted entities in Cosmos DB.

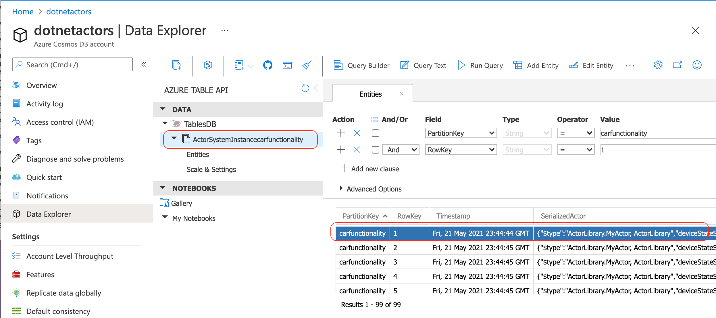


Fig23. Cosmos DB showing persisted Actors

Fig24 shows car sample code running with command line argument *shallRun* as false.

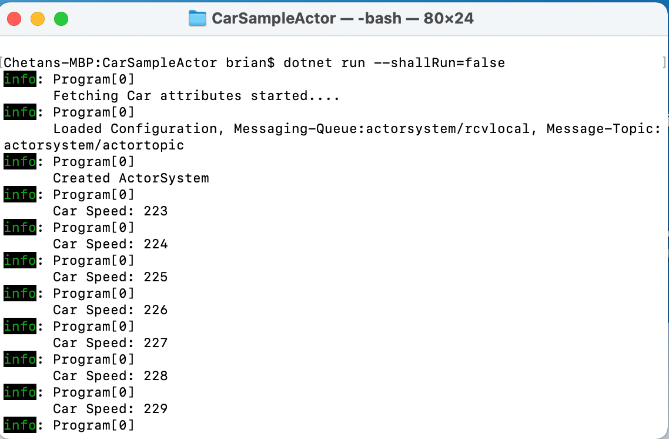


Fig24. Persistence logs in-car sample code

##### References

1. K.K.Tiwari, “Actor Model in Nutshel“ , Jan 28, 2019 , Available [Online]:

<https://medium.com/@KtheAgent/actor-model-in-nutshell-d13c0f81c8c7#:~:text=Actor%20Model%20is%20a%20conceptual%20concurrent%20computation%20model%2C%20came%20into,Actor%20Model%20are%20Akka%20%26%20Erlang>.

1. Akka.net, “Actors“ , Available[Online]: <https://getakka.net/articles/concepts/actors.html>
2. B.Storti, July 9, 2015. Available [Online]: <https://www.brianstorti.com/the-actor-model/>
3. Microsoft, “Azure Cosmos DB”. Available [Online]

<https://docs.microsoft.com/en-us/azure/cosmos-db/introduction>

1. Microsoft, “Azure Service Bus”. Available [Online]

<https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview>

1. Microsoft, “Deploy and run web app with Azure app service”. Available [Online]

https://docs.microsoft.com/en- us/learn/modules/deploy-run-container-app-service/